14. The UTIL Module

Created: April 1, 2003
Updated: September 16, 2003

The UTIL API [Library **xutil**: include | src]

- Chapter Outline

The UTIL module is a collection of useful classes which can be used in more then one application. This chapter provides reference material for many of UTIL's facilities. For an overview of the UTIL module please refer to the UTIL section in the introductory chapter on the C++ Toolkit. The following is an outline of the topics presented in this chapter:

- Miscellaneous Classes
    - class CLightString
    - class CChecksum
- Input/Output Utility Classes
    - class CIStreamBuffer
    - class COStreamBuffer
    - class CByteSource
    - class CStreamByteSource
    - class CFStreamByteSource
    - class CFileByteSource
    - class CMemoryByteSource
    - class CByteSourceReader
    - class CSubSourceCollector

Test Cases [src/util/test]

Containers

The Container classes are template classes that provide many useful container types. The template parameter refers to the types of objects whose collection is being described. An overview of some of the container classes is presented in the introductory chapter on the C++ Toolkit.

The following classes are described in this section:

- template<typename Object> class CWeakMapKey
- template<typename Object> class CWeakMap
- template<typename Coordinate> class CRange
- template<typename Object, typename Coordinate = int> class CRangeMap
- template<typename Object, typename Coordinate = int> class CRangeMultiMap
- class CIntervalTree

template<typename Object> class CWeakMapKey

```
#include <util/weakmap.hpp>
```

This class is used in conjunction with CWeakMap.

template<typename Object> class CWeakMap

```
#include <util/weakmap.hpp>
```

This class is used in conjunction with CWeakMapKey. It is extension for regular maps with additional feature: it automatically removes elements from map when corresponding key is destructed. Key is of type CWeakMapKey<Object>.

```
//  Generic example of usage of these templates:
#include <util/weakmap.hpp>

class CKey
{
public:
    CWeakMapKey<string> m_MapKey;
};

void Test(void)
{
    // declare map object
    CWeakMap<string> map;
    {
        // declare temporary key object
        CKey key;
        // insert string value
        map.insert(key.m_MapKey, "value");
        cout << map.size();
        // == 1
        cout << map.empty();
        // == false
    } // end of block: key object is destructed and map forgets about value
    cout << map.size(); // == 0
    cout << map.empty(); // == true
};
```

Typedefs

```
key_type
mapped_type
value_type
iterator
const_iterator
```

are the same as in standard C++ template map<>.

Methods

```
size_t size() const;
bool empty() const;

const_iterator begin() const;
const_iterator end() const;
const_iterator find() const;

iterator begin();
iterator end();
iterator find();
```

are the same as in standard C++ template map<>.

```
void insert(key_type& key, const mapped_type& value);
void erase(key_type& key);
```

do the same as corresponding methods of standard C++ template map<>. They differ only in return type.

template<typename Coordinate> class CRange

Class for storing information about some interval (from:to). From and to points are inclusive.

Typedefs

```
position_type
```

synonym of Coordinate.

Methods

```
CRange();
CRange(position_type from, position_type to);
```

constructors

```
static position_type GetEmptyFrom();
static position_type GetEmptyTo();
static position_type GetWholeFrom();
static position_type GetWholeTo();
```

get special coordinate values

```
static CRange<position_type> GetEmpty();
static CRange<position_type> GetWhole();
```

get special interval objects

```
bool HaveEmptyBound() const;
```

check if any bound have special 'empty' value

```
bool HaveInfiniteBound() const;
```

check if any bound have special 'whole' value

```
bool Empty() const;
```

check if interval is empty (any bound have special 'empty' value or left bound greater then right bound)

```
bool Regular() const;
```

check if interval's bounds are not special and length is positive

```
position_type GetFrom() const;
position_type GetTo() const;
position_type GetLength() const;
```

get parameters of interval

```
CRange<position_type>& SetFrom();
CRange<position_type>& SetTo();
```

set bounds of interval

```
CRange<position_type>& SetLength();
```

set length of interval leaving left bound (from) unchanged

```
CRange<position_type>& SetLengthDown();
```

set length of interval leaving right bound (to) unchanged

```
bool IntersectingWith(CRange<position_type> range) const;
```

check if non empty intervals intersect

```
bool IntersectingWithPossiblyEmpty(CRange<position_type> range) const;
```

check if intervals intersect

template<typename Object, typename Coordinate = int> class CRangeMap

Class for storing and retrieving data using interval as key. Also allows efficient iteration over intervals intersecting with specified interval. Time of iteration is proportional to amount of intervals produced by iterator. In some cases, algorithm is not so efficient and may slowdown.

template<typename Object, typename Coordinate = int> class CRangeMultiMap

Almost the same as CRangeMap but allows several values have the same key interval.

class CIntervalTree

Class with the same functionality as CRangeMap but using different algorithm. It is faster and its speed is not affected by type of data but it uses more memory (triple as CRangeMap) and, as a result, less efficient when amount of interval in set is quite big. It uses about 140 bytes per interval for 64 bit program so you can calculate if CIntervalTree is acceptable. For example, it becomes less efficient than CRangeMap when total memory becomes greater than processor cache.

Thread Pools

This section provides reference to the classes used to implement a pool of threads. For an introduction to this topic, see the Thread Pools section in the introductory chapter on the C++ Toolkit.

The following classes are discussed in this section:

- template <typename TRequest> class CBlockingQueue
- class CStdRequest
- template <typename TRequest> class CThreadInPool
- class CStdThreadInPool
- template <typename TRequest> class CPoolOfThreads
- class CStdPoolOfThreads

template <typename TRequest> class CBlockingQueue (%20)

A blocking queue is a first-in-first-out container with the special property that attempting to extract an element from an empty queue blocks efficiently until more elements are available. Thread pools use this class internally to manage requests.

class CStdRequest (%20)

Abstract class, derived from CObject, encapsulating requests to aCStdPoolOfThreads. The pure virtual method ***void Process(void)*** gets called when a thread handles the request.

template <typename TRequest> class CThreadInPool (%20)

Abstract class, derived from CThread, for the threads in a pool. Three virtual methods control its behavior:

```
virtual void Init(void) {} // called at beginning of Main()

// Called from Main() for each request this thread handles
virtual void ProcessRequest(const TRequest& req) = 0;

virtual void x_OnExit(void) {} // called by OnExit()
```

class CStdThreadInPool (%20)

A specialization of ***CThreadInPool*** for CRef<CStdRequest>, which simply processes each request by calling its ***Process()*** method.

template <typename TRequest> class CPoolOfThreads (%20)

Abstract class for a pool of request-handling threads. The constructor takes three arguments: the maximum size of the pool, the maximum size of the queue of pending requests, and an optional threshold (compared to the difference between the number of unfinished requests and the number of threads in the pool) indicating when to create another thread automatically. Due to some limitations of C++, the constructor does not create any threads itself; you have to call ***Spawn()*** for that. In addition, ***AcceptRequest()*** passes requests to the pool, and the protected pure virtual function ***NewThread()*** creates the actual threads.

class CStdPoolOfThreads (%20)

A specialization of ***CPoolOfThreads*** for CRef<CStdRequest>; its ***NewThread()*** method creates objects of class CStdThreadInPool. It also introduces a new method:***KillAllThreads()***, which causes all the threads in the pool to exit cleanly after finishing all pending requests, and takes an argument indicating whether to return immediately or to wait for them to finish.

Miscellaneous Classes

The following classes are discussed in this section. For an overview of these classes see the Lightweight Strings and the Checksum sections in the introductory chapter on the C++ Toolkit.

- class CLightString
- class CChecksum

class CLightString

Class for storing information about char strings. Unlike standard C++ string class it doesn't take ownership over string contents. So, char array containing string value should exist for whole life of holding CLightString object. This char array should be deleted (if needed) after CLightString object destruction by some other mechanism. Note that for efficiency sort order of CLightString differs from standard. It compares first by string length, then by string contents, so here is an example of sorted data:

```
"a"
"z"
"aa"
"az"
"zz"
"aaa"
```

which, if sorted by standard comparison will look like:

```
"a"
"aa"
"aaa"
```

```
"az"
"z"
"zz"
```

class CChecksum

Class for CRC32 checksum calculation. It also have methods for adding and checking checkum line in text files.

Input / Output Utility Classes

This section provides reference information on a number of Input/Output Utility classes. For an overview of these classes see the Stream Support section in the introductory chapter on the C++ Toolkit.

- class CIStreamBuffer
- class COStreamBuffer
- class CByteSource
- class CStreamByteSource
- class CFStreamByteSource
- class CFileByteSource
- class CMemoryByteSource
- class CByteSourceReader
- class CSubSourceCollector

class CIStreamBuffer

Class for additional buffering of standard C++ input streams (sometimes standard C++ iostreams performance quite bad). Uses CByteSource as data source.

class COStreamBuffer

Class for additional buffering of standard C++ output streams (sometimes standard C++ iostreams performance quite bad).

class CByteSource

Abstract class for abstract source of byte data (file, stream, memory etc).

class CStreamByteSource

CByteSource subclass for reading from C++ istream.

class CFStreamByteSource

CByteSource subclass for reading from C++ ifstream.

class CFileByteSource

CByteSource subclass for reading from named file.

class CMemoryByteSource

CByteSource subclass for reading from memory buffer.

class CByteSourceReader

Abstract class for reading data from CByteSource.

class CSubSourceCollector

Abstract class for obtaining piece of CByteSource as separate source.